A Survey of Concurrent Object-Oriented Programming Languages Technical Report RM/96/4

N. R. Scaife
Dept. of Computing and Electrical Engineering
Heriot-Watt University
Edinburgh EH14 4AS

February 7, 1996

Abstract

Issues in the design and selection of concurrent object-oriented languages are discussed and some general principles outlined. The bulk of this paper is an extensive (although far from complete) enumeration of concurrent object-oriented languages, past and present. Salient features are discussed and unusual or novel aspects of each language pointed out. An attempt is made to draw some general conclusions from work in this area in terms of the features provided by the languages and the hardware they are targetted on.

1 Motivation

We are currently engaged in a project to exploit functional programming as a means of developing and maintaining parallel programmes. To date we have developed a number of hand-built applications in the area of computer vision [118, 88, 99] and have developed a prototype automatic parallelising compiler based on Standard ML [24]. We now wish to investigate the role of a higher-level abstraction mechanism in our methodology and would like to base our research in the light of existing projects. Although the search for abstraction mechanisms in parallel programming has been going on for a long time, it has proved to be a much more difficult task for parallel systems than for sequential systems, for which abstraction plays a major role in most modern sequential languages.

One vibrant area of research which has arisen in the last eight years, or so, is the idea of using object-oriented code as the basis for parallel programming. This has spawned a large number of different languages, systems and methodologies and has proved tractable to theoretical analysis. Some of these languages implement abstraction mechanisms very similar to the techniques we employ in developing parallel code and we would like to draw on this experience in our endeavours. We have gathered some of the available literature and attempted to look for general principles in this work.

Here, we present a summary of the languages discovered, 29 programming languages plus a number of operating systems and even an abstract machine. We do not project the relevance of this survey onto our intended investigation, which, after all, will be based on functional abstract data types rather than persistent objects but we draw some tentative and general conclusions from our findings.

2 Concurrent Object-Oriented Programming

2.1 Introduction

In sequential languages, the procedure or function is the basic abstraction mechanism, code and data can be combined and parameterised in such a way that they can be reused with suitable instantiation. This does not extend to parallel programming, however, since it does not adequately describe the communication and synchronisation required between concurrent objects [6].

The problem with parallel computation is that a high degree of *indeterminacy* is introduced by the fact that different segments of a parallel computation can be distributed throughout *space* and are executed at indeterminate points in *time* relative to one another [3]. These issues are *transparent* in a sequential execution. One consequence is that partially evaluated sections of programme coexist which greatly complicates the location and synchronisation of these sections and increases the risk of harmful interference between them.

Parallel programming thus requires a new order of abstraction mechanism whereby not only are programme instructions and data abstracted but the location of data and the pattern of computation are *separately* abstracted and parameterised. Ideally these two should be independent and orthogonal to one another, in the sense that different patterns of computation may be required for the same sequential abstraction on different parallel architectures, however, this has proved an extremely difficult goal to achieve.

This is one of the strengths of applying Object-Oriented programming techniques to parallel programme development. The management of potentially parallel programmes where communication and synchronisation are required are succinctly modelled by objects in an object-oriented environment. The objects encapsulate the code and data sections to be executed in parallel and the interaction between objects corresponds to the evaluation strategy.

This creates a hierarchical view of the overall process of specifying and developing parallel programmes [6]:-

1. At the lowest level are the basic concurrency primitives. These involve communications primitives such as sending messages and specifying filters for incoming messages. Also included here are computation primitives such as specifying concurrent execution either remotely or locally. These will be present in the language specifying an object-oriented solution to a programming problem but will also be mapped through an intermediate representation (if one is used) and will map onto the native language of the target hardware.

2. Above this layer are the patterns of communication that connect objects together. These can be, for instance:-
   (a) Point-to-point as in the remote procedure call (RPC) methodology. This is a subset of the general message passing concept.
   (b) Pattern directed in which groups of anonymous recipients can be addressed. The actual pattern of messages can be abstracted and defined by the message transmitter.
   (c) Constraint-based where message recipients can filter out specific classes of messages. These could be to specify specific orderings of events, for instance reading from an empty buffer, which are difficult to setup in asynchronous systems.

3. At the top level is a *modular decomposition* [6] which is a top-level abstraction layer used by the designer to separate design concerns from implementation issues. For instance, this includes abstracting away from placements of tasks onto processors, which is in general an intractable problem [23], but for which efficient solutions for particular cases can be parameterised and reused.

As with parallel functional languages there is a wealth of parallelism contained within an object-oriented programme:-

1. Individual objects, or groups of objects, could be realised on physically remote processing elements and executed in parallel. This is a natural consequence of modelling programme entities as real-world objects, which co-exist as do physical objects in reality.

2. A single object, however, may itself be spread over several processing elements executing methods in parallel and, in fact, an extreme of this would be to have *all* programme objects spread over *all* of the processing elements.

3. Furthermore, each object's method could itself be executed in parallel with groups of processing elements allocated to each method, or shared between methods.

A great deal of the effort in parallel functional languages has gone into controlling this explosion of parallelism and expressing characterisable parallel constructs in a reusable form. This also applies to concurrent object-oriented technology where the object-oriented features allow the parallelism in a programme to be expressed and also provide the mechanisms for constraining and controlling the parallelism.

One consequence of this is that the parallelism in a COOP language should be centred around the objects envisaged by the programmer and not simply expressed in the host language with the object-oriented features providing a higher-level description of the overall structure of the programme, which is roughly what it does in sequential object-oriented programming. This precludes the third type of parallelism indicated above.

2.2 Can OOP Help in Constructing Parallel Programming Problems?

The advantages of object-oriented methodologies in sequential programming are well documented [38, 84, 71, 64]. The difficulties associated with parallel programming are also well known and the basic idea is to apply object-oriented methodologies to the engineering of parallel systems [80].

The idea is that object-oriented programmes model things which exist (concurrently) in the real world and as such aid programmers in thinking about parallel programmes which have concurrent entities. Nobody seems to have attempted to justify this, however, by proving that an object-oriented decomposition of a problem is necessarily an efficient one for parallel execution.

The basic problem is to retain the advantages of conventional object-oriented methods while providing efficient and flexible control over parallelism and providing a separate level of abstraction for parallel constructs. It is relatively easy to implement message passing parallelism in object-based systems but this loses some of the power and generality of object-oriented culture. The problem is that some of the features of a state of the art object-oriented language are difficult to implement in a parallel system, particularly inheritance although inheritance also has recognised problems for sequential systems [81] which require language support to overcome [104].

There are a number of issues which have to be addressed [111, 93] before an object-oriented language or system can be uniformly applied to development of parallel applications. This could be direct in the sense that the object-oriented language is the host language for the application or indirect meaning the application of object-oriented methods in the design, with possible automatic translation into the native language of the target architecture.

2.2.1 Sharing of Variables

In object-based languages, all variables are local to the objects and no shared variable conflicts arise. When inheritance is implemented, however, *class variables* can be accessed by all the instances of the class to which they belong. This creates a problem in that instances of a given class cannot be guaranteed to be at the same physical location and updating one requires updating messages to be sent to all the other instances, which can result in consistency problems.

Two solutions have been proposed [93].

1. Some form of mutual exclusion could be implemented so that only one instance can update the variable at any time. This is apparently an uncommon solution to the problem [112].

2. Another approach is to treat the classes themselves as objects. In this case an instance of a class has to request the parent class object for access to class variables.

Instance variables are almost universally accepted as belonging to the instance object and all accesses to these variables are through the objects' methods. One complication is in systems where the methods themselves are implemented concurrently and the same problems of consistency arise with variables shared within the object between concurrent methods.

2.2.2 Communication models

There are a number of ways of implementing message passing between concurrent objects, based on whether sends and receives are *blocking* or *non-blocking*. Figure 1 gives event diagrams of some of the most common methods in use.

If the send is non-blocking and the receive is blocking then *asynchronous message passing* is the result. The actor system [56, 2] and one implementation of Concurrent Smalltalk [41] use this type of message.

Synchronous message passing is when there is a blocking send and a blocking receive. Hoare's Communicating Sequential Processes (CSP) [60, 58] uses this type of message. Since the synchronous and

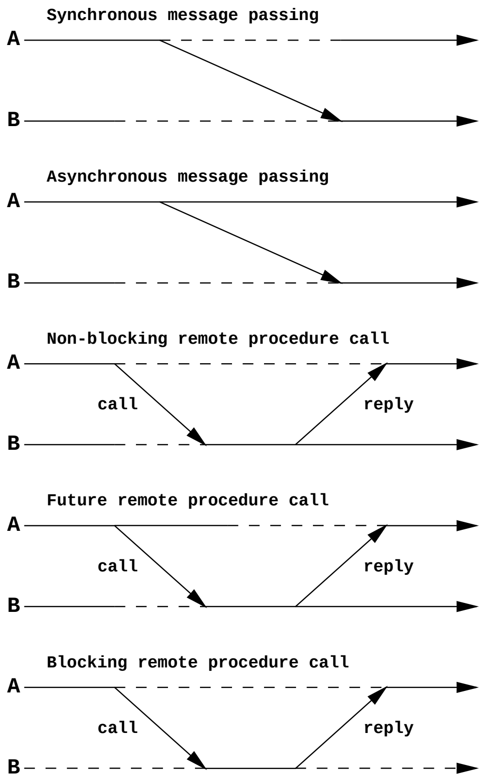

Figure 1: Modes of operation for message passing systems

asynchronous methods can emulate each other [112] the two are identical up to the types of algorithms they can express. One characteristic of synchronous message passing, however, is that no buffering is required and most asynchronous systems implement some kind of blocking when buffer space runs out.

If message passing is implemented using a *call* and *reply* system then this is the *Remote Procedure Call* (RPC) system and there are several variations on this theme.

1. *Non-blocking* RPC is when the caller is blocked until it receives a reply from the replying node. The target node is blocked until the reply is sent.

2. *Future* RPC is similar to non-blocking RPC except that the caller is not blocked until the reply becomes necessary for further progress. If the reply is received before this point then the caller is not blocked at all.

3. *Blocking* RPC is the traditional view of remote procedure call and the target node is only active when it is processing a call from the calling node. Concurrency only arises when each node can service more than one caller. This has been popular because it preserves the semantics of sequential code and can be used to upgrade sequential code to run on a concurrent system.

Some systems allow a choice between some of these models (*selective waiting*) [28]. Another enhancement is the prioritisation of messages, for instance ABCL/1 [26] implements *express* and *ordinary* messages whereby an express message can interrupt the processing of an ordinary message.

2.2.3 Distributed object management

A number of issues arise in managing concurrent objects [42].

1. *Object naming.* Each object must have a unique name within the system so it can be addressed by other objects and by system management. It is desirable for this to be independent of the location of the object to allow dynamic placement of objects during operation.

2. *Object location.* The static location of objects is simply the *mapping problem* in another form [23], although mapping objects as opposed to tasks is slightly easier due the larger size of the objects being placed [89]. There is also the potential for controlling the complexity of this calculation by controlling the sizes of the objects.

   One of the advantages of having concurrent objects, however, is the potential for programme optimisation by relocation during execution. This can be to balance the workload on each processing element or to minimise communication costs by ensuring communicating objects are located at the same physical location.

3. *Object size.* This controls the *granularity* of the system and can range from very fine-grained such as the actor model or much coarser which matches multiprocessor systems.

4. *Object type.* Some concurrent object-oriented systems define several different types of objects, for instance CHARM++ [67] implements *concurrent*, *replicated*, *shared* and *communication* objects. Other systems implement *fixed* and *mobile* objects.

The question of object location is a thorny one, especially in systems implementing inheritance where *class consistency* has to be maintained. If the entire class hierarchy is guaranteed to be on the same processing element then no problems arise. If, however, class instances migrate independently of the class objects then class variables have to be maintained across processor boundaries with the attendant potential for errors. If the object to be migrated is to be read-only then some systems implement a cloning system whereby copies of a remote object are made for local access.

Another issue in object location is that of *transparency*, in other words, are callers aware of whether their calls are serviced locally or remotely? One method of achieving this is to implement a virtual naming system in which case a separate mechanism is required to control objects' locations. Another method involves *proxies* [42]. A proxy is a dummy object on one site which receives access calls locally and then translates these local calls into remote accesses to the actual object.

2.3 Modelling Object Behaviour

2.3.1 Actors

The **actor** model was first proposed by Hewitt [56] and further developed by Agha [2]. The method has proved tractable to formal treatment and proof theory [8] and has spawned an important class of programming languages such as HAL [61], the ABCL [26, 119] series of languages and Cantor [15] and even systems support such as Rosette [111] and the J-machine [40].

The actor methodology consists of independent, asynchronously communicating objects for which three basic primitives are provided:-

create is the equivalent of lambda abstraction in the functional paradigm but extends the dynamic resource allocation provided by function abstraction.

send-to is the asynchronous analogue of function application or procedure invocation in procedural languages. Each actor has a unique mail address and a mail queue.

become allows modelling of mutable shared objects and gives the history-dependent behaviour needed by such objects.

Change in state is implemented by a replacement behaviour strategy, this can be simple such as a change in variable or higher level such as a change in the object's methods. Replacement behaviour has important implications for deciding when unit operations have completed. Unlike assignment statements the granularity of the concurrency is not fixed by the units of operation allowing greater parallelism to be exposed while giving tight control over the various threads of computation and programme states that occur during a concurrent execution.

Actors are extremely low-level entities, for example numbers in the actors methodology are represented by an actor with the value of that number. Actors are divided up into *primitive* and *non-primitive* actors, where primitive actors such as numbers have no mail-address and are simply identified by their value and act immediately upon any messages that are sent to them, for instance sending the message **[+4]** to the actor **3**. Primitive actors take the place of atomic entities in a conventional programming paradigm and are intended to prevent infinite sequences of message passing [111].

Another characteristic of actors is that there are two assumptions implicit in the execution model [75]:

1. All messages are guaranteed to be received within a bounded time-interval.

2. An actor waiting to execute will eventually do so.

There are numerous experimental languages based on the actor methodology, although some of them do not implement the replacement strategy specified in the pure actor definition, inasmuch as actors persist between message receptions.

The actors methodology has been a very strong thread in concurrent object-oriented research. Actor languages have allowed a high degree of abstraction to be built into parallel applications,for instance, *ActorSpace* [5] is a system of programming that allows abstraction over groups of uniform actors. Limited semantics preserving transformations on objects are also possible, for instance, *join continuations* allow transformation of RPC-style communications into asynchronous equivalents [7]. The creation of these high-level abstractions also opens up interesting possibilities in terms of dynamic control of parallelism, for instance *reflection* is when an application has access to a model of itself which it can update as the computation proceeds [82, 4].

2.3.2 Message passing

The concept of objects communicating by messages is the most general of the methodologies given here, both actors and remote procedure calls are subsets of this methodology, for instance the *Message Driven Computing* (MDC) model [36] is a generalisation of the actors model.

There are, however, a number of different ways of implementing this idea, with issues such as *synchronous* versus *asynchronous* message passing, *buffered* versus *unbuffered* messages, granularity of messages, the object *synchronisation* mechanism (as opposed to the message synchronisation mechanism), the *activation* and *deactivation* of processes, the *identification* of nodes and processes.

With all these choices to be made there are a vast number of possibilities for basing languages and systems on this method, some even implement several different varieties of message management in the same system, for example [30] introduces a system which has both passive (*non-process*) objects which only act when they are sent messages and active (*process*) objects which have an independent existence and can process independently of other objects. Process objects are communicated using asynchronous messages whereas non-process objects service RPC style calls. This method is interesting in that it lies somewhere in between pure message passing and RPC, generally RPC is considered *safer* and message passing more *powerful*.

2.3.3 Remote procedure call

The Remote Procedure Call paradigm has been a very influential concept in the design and implementation of distributed operating systems [106]. The basic idea is very simple, that is to make remote system access appear as similar to the access of local resources as possible. There are, however, a number of subtle issues in the design of an RPC system which complicate matters [101], for instance RPC calls may be broadcast to all nodes instead of point-to-point, some RPC calls may not necessarily expect a reply, they can be blocking or non-blocking and there are implementational issues such as timeouts for non-returning calls, pure RPC will wait indefinitely but this could be a problem in a distributed system. An RPC design is, therefore, a trade-off between several conflicting system parameters to get adequate performance.

RPC calls can also be classified according to the call semantics, [103] divides reference semantics under failure conditions into *maybe*, *at-least-once*, *only-once-type-1* and *only-once-type-2* and discusses their behaviour under different failure conditions. In addition there are *request-response* and *request-response-acknowledge* protocols.

When adapting this idea for parallel systems, the consequences of failures are not, in general, considered although these can lead to situations such as deadlock. A number of parallel languages use RPC or some variant thereof as their execution model, for instance Emerald [65], Mentat [49] and CC++ [33] although most of these allow a fair degree of flexibility in the interpretation of the RPC method. In general, RPC is considered slower than general message passing but allows existing sequential code to be re-engineered in parallel much easier.

3 Concurrent Object-Oriented Languages

The following is an extensive although incomplete summary of many of the attempts to produce a concurrent object-oriented language, organised *very* roughly into chronological order. Some languages are highly experimental whereas others are almost production quality and it is likely that many of them are no longer under development. The idea behind this list is to give an overall impression of the way developments and lines of enquiry have progressed in this popular area of endeavour and unfortunately there is a high degree of repetition between many of them so some effort has been made to concentrate on the unusual or advanced features of each one.

BETA [72] is a language that has been developed since 1976 and is a SIMULA [39] based language but generalises some of the concepts introduced by SIMULA. The idea behind BETA was as a small, compact but highly expressive language that could be used to develop large scale code, including distributed systems.

The basic abstraction mechanism is the *pattern* which replaces *classes*, *procedures*, *functions* and *types*. Instances of patterns are called *objects* and can be used as variables, data structures, procedures, functions, coroutines and concurrent systems. The basic execution mechanism is called the *evaluation* which causes a change in state and produces a value.

BETA objects can be; *system* which execute concurrently, *component* which can be alternated with other components and *item* which is a sequential action dependent on other objects. In addition, items can be *static*, *inserted* (which fits between two patterns) or *dynamic*. Inheritance is implemented using the *superpattern* mechanism. This includes explicit control of overriding using the *inner* construct. The SIMULA notion of virtual procedures is generalised to *virtual patterns* whereby the full specification of a pattern is not required when it is defined.

Concurrency is expressed using three methods of organising action sequences; *sequential*, *alternation* and *concurrency*. Concurrent objects communicate through synchronised execution of items, similar to an Ada *rendezvous*, the sending object requests that the receiving object execute and item and the sending object names the receiving object but the receiver can place conditions on the type of sending object using the *some* construct. This system is intermediate between the CSP method where both sender and receiver must be named and the Ada system where receivers accept all communications. Other types of communication are provided such as asynchronous communication between internal systems (subclasses in a hierarchy) and communication through global objects.

CLIX [62] is is a language which operates in a manner somewhat similar to the actors model in that objects exist concurrently and communicate through message passing. Objects are only active when in receipt of a message which can be both synchronous or asynchronous, the underlying model is of a mail system based on system-wide unique *object-id*s. The motivation here, however, is that each object encapsulates not only its procedural behaviour but also its communications handler which is invariant over the lifetime of the object.

The communications system guarantees that message ordering between two objects is preserved which gives a slightly simpler semantics than the actors model. Another characteristic is that mail messages have unique ids and contain a *reply-to* field which tells the recipient where to forward any replies.

Inheritance is implemented as classes of objects with common behaviour. There is no structural relationship among classes, instead, inheritance is implemented by *delegation* [79] where if an object receives a request is cannot service it forwards the message to an object which can.

Concurrent C++ [46] was based on the concept of combining two existing extensions to the C language; C++ which implements data abstraction and Concurrent C which can be used to develop parallel systems. Combining two orthogonal components such as parallelism and data abstraction requires extended semantics and there are a number of issues that have to be addressed.

This language was an early attempt at a concurrent C++ and used simple mechanisms to merge facilities from each language, for instance class variables can be used from within process bodies and the constructors and destructors are called on entry and exit. Little effort was made to address locality and remote access issues, for instance reference types could not be used unless the system had some form of shared memory.

Several problems were highlighted but no solutions offered. An example of this is when an object is terminated by a process that did not create it, the creating process has to be informed of the change but would necessitate additional communication. Another example is the calling of destructors when processes terminate in such a manner that class consistency is retained. Note also that problems with inheritance were hinted at but no in depth analysis of this was given.

Even in this simple system, however, many useful features were discovered, such as using an *interface class* to aggregate a group of classes with a single interface.

Emerald [22, 65] is a language and system designed to exploit both fine- and coarse-grained object mobility in an object based system. Remote calls are implemented in an RPC-like manner but with some additional facilities to decide when an object should be moved or not.

There are explicit facilities for locating and moving objects such as *Move*, *Fix* and *Unfix*. Additionally some decisions are made by the compiler such as for parameter passing, small immutable objects are always are copied to the remote site for remote invocation and there is a *call-by-move* parameter passing method that also defines copying to the remote site on invocation.

Objects can be *global* system wide objects, *local* objects contained within other objects and *direct* which are for primitive types such as integers. Global object ids are kept track of using a *forwarding address* mechanism so each object has a timestamped record of where all global objects are located. A broadcast scheme is used when an access fails to locate an object. Objects can be kept together on the system by declaring *attached* variables so that when an object is moved, objects to which it is attached move with it.

Emerald provides the simplicity of object-based programming which does not have some of the complexities of *process* mobility used by some systems. It also combines the advantages of fine-grained mobility for data access and coarse-grain mobility for load balancing, remote data access, error recovery and specific hardware access (eg. disc, screen).

PRESTO [20] is the predecessor of the Amber [34] system and is based on C++. Objects in PRESTO encapsulate not only data and implementation but also an *execution* behaviour, which is claimed make parallel programming simpler. Remote access is also transparent which allows the system implementor to choose between parallel or sequential execution, although the user can choose between synchronous or asynchronous communication.

There are several classes which express and control concurrency. The *thread* class defines execution threads and can exist concurrently. The *synchronisation* class defines *relinquishing* and *non-relinquishing locks* for synchronisation between concurrent threads (the latter are for immutable objects such as hardware). More sophisticated synchronisation is provided through *monitors* and *condition variables* [59].

One of the goals of PRESTO is to provide the programmer with the ability to define parallel constructs and execution mechanisms in a manner that allow reuse of parallel code. To what extent this has been achieved is unclear.

SR (Synchronizing Resources) [12] is a language designed to develop distributed systems, influenced by Modula-2 and CSP, and like Modula-2 is a development of an earlier language (SR_0) and is based on experience with that language. SR is based on *resources* and *operations*, resources encapsulate code and data and operations are the mechanism for process interaction. Inheritance is also supported. A multitude of communications paradigms are provided including RPC, rendezvous, asynchronous message passing, multicast and semaphores. A uniform representation for sequential and concurrent constructs is used, similar to CSP and there is only one abstraction mechanism for both, the resource.

Acore [83] is based on both the actors model and on lambda calculus (Lisp). The idea, here, is that lambda calculus provides a convenient syntax for expressing concurrency whereas the actor model gives a model of concurrently existing objects with a state and communicating through message passing. These are combined by viewing lambda expressions as patterns of message passing, applying a function to its arguments constitutes a *request* message and the return value is associated with a *reply* message. Applicative order evaluation is used by default so sub-expressions are evaluated prior to issuing messages, which may themselves require messages to be sent, hence a lambda expression defines a particular sequence of messages. Lazy and strict evaluation are also possible.

ACT++ [66] makes use of a perceived relationship between the techniques used in concurrent object-oriented languages and real-time systems. The main intention is to develop concurrent real-time systems using a concurrent object-oriented language with support from a distributed run-time kernel called REACT. ACT++ is based on C++ with actor semantics augmented with inheritance. Each actor is implemented at the procedure level which defines fine to medium grained concurrency. The REACT kernel is based on CHOICES [29].

Actra [107] is an implementation of Smalltalk-80 [48] which uses actor semantics to extend Smalltalk for multiprocessing applications. A new class of objects called *Actors* is defined which behave as normal Smalltalk objects (they can send and receive messages) but can be executed in parallel on separate processors. Synchronous send and receives are implemented along with a non-blocking *reply:* message.

Amber [34] is a development of Presto [20] and is implemented as a pre-processor for an object-based subset of C++ combined with a run-time kernel. The target hardware is networks of shared memory multicomputers and the system executes as a uniform system-wide object space. The main priority seems to be the parallel performance so object placement is under programme control rather than run-time control. Unlike other object-based systems Amber is intended to run a single algorithm and terminate so support for object persistence and communications outside the programme are not included.

Programme execution proceeds as lightweight threads and all threads that manipulate a given object are contained on a single node. Object consistency is maintained by hardware synchronisation. The programmer interface is a set of object classes for controlling threads, synchronisation and object distribution. Threads can be dynamically created and are based on Presto's thread management. Synchronisation classes are also provided and give the same facilities as Presto but are extensible to providing user defined synchronisation mechanisms. Since threads follow the objects they act on, communications are controlled by object location. Object location is based on Emerald's [65] system but are specified at run-time rather than compile time.

Blaze 2 [87] is an object-oriented language that concentrates on highly parallel access to shared objects. There are two ways of preventing multiple threads accessing an object from interfering with each other. Either access is limited to one method at a time (this is used by, for instance, ABCL/1 [26]), or synchronised access to shared variables are required which is what Blaze 2 provides. Shared variables can be locked out under user control when accessed allowing deterministic access to shared objects.

Plasma-II [75] is a development of Plasma which was one of the first actor-based languages. This extension has more explicit control of concurrency, two types of actors are defined; *pure* actors which behave in the normal actor manner and *serialised* actors which allows sequential processing of messages to allow a shared resource controller to be expressed. Four types of transmissions are defined, each combination of blocking/non-blocking and sequential/parallel. Actors can also be grouped together either sequentially or in parallel and broadcasting is defined in several varieties, including futures.

A'UM [122, 123] is unusual in that it is based on stream semantics [74]. This gives a declarative system which is somewhat similar to the actors model but with explicit control over the order of messages and is relational rather than functional. It is designed to overcome some of the problems with concurrent object-oriented logic programming. The computational model is of the production and consumption of streams, termination is when there are no messages left. This integrates well with objects which are viewed as producers and consumers of streams although streams can themselves be looked upon as objects. A'UM gives an elegant and sound framework but looks somewhat difficult to programme in.

ABCL/1 [119] (Actor Based Concurrent Language) was an early variant in the ABCL series of languages but has been substantially developed since then [120]. Three types of message are defined; *past* which is asynchronous, *now* which is synchronous and *future*. There is also a simple two level prioritisation of messages *ordinary* and *express* messages. The actors component can also coexist with traditional programming concepts such as procedural or functional paradigms (Lisp).

ABCL/1 is an object-based concurrent language, remote access to objects is controlled by a system of local and global identifiers, the local identifier is for an entry in a local object table which contains the current location of the object. Both explicit and automatic placement of objects is provided. ABCL/1 was designed to operate at a higher level of granularity than most actor based languages to ease the problems of object/processor allocation.

Recent developments in ABCL include an implementation on EM-4 [120] which is a precursor to the massively parallel machine EM-X, defined as part of Japan's *Real World Computing (RWC)* programme (the successor to the Fifth Generation Computing Project).

C++ Parmacs [19] is an extension of the M4 parallel programming macros for C (called Parmacs). Parmacs allows development of shared and distributed memory machines and is based on monitors supported by five basic primitives; *basic lock*, *simple monitor*, *barrier* (a synchronisation mechanism), *getsub* (an atomic parallel loop) and the *askfor monitor* which manages a queue of processes. C++ Parmacs turns these primitives into classes in a hierarchy with the `class BasicLock` as the lowest level class and allows new classes to be derived from the basic classes by inheritance, `class LeakLast` and `class LeakOne` are barriers that allows one process to execute while the others are blocked, `class simpleQ` and `class generalQ` are subclasses of askfor based on lists of processes. Few of the inherent problems with concurrent object-oriented programming languages are mentioned in the implementation, this may be due to the use of monitors as the concurrency primitive but if this language were to be extended to support multiple threads, as suggested [19], then solutions to these problems may be required.

POOL [13] is a family of object-oriented languages designed to support the DOOM architecture which is a network of distributed memory processors, specifically designed to support concurrent objects. Objects exists concurrently and use the Ada rendezvous style of communications. Inheritance and subtyping were considered for POOL but it was, rather strangely, concluded that inheritance was less relevant for parallel object-oriented languages than for sequential ones [10].

Lisp has been used to develop object-oriented programmes and [97] presents an attempt to re-engineer parallel Lisp programmes from sequential ones using predefined concurrency primitives. Three concurrency primitives are used at different levels of granularity. *Futures* [52] where the argument to a function is executed in parallel with the function itself (the function blocks if it uses the value of the argument before it has been evaluated). *Linda* [32] is a method whereby commonly used operations are given a distinct parallel execution and exist concurrently with the main programme. A means of controlling multiple access to a given operation is required. *Time-warp* [63] is a speculative evaluation model. Computations are carried out before they are needed (or even whether they are needed at all). The problem with imperative systems is that of *rollback* where side-effects have to be undone if the computation is not required, this is not a problem with functional execution.

This implementation of Lisp implements process migration for time-warps and futures. Both manual parallelisation and semi-automatic parallelisation are supported. Static estimation of parallel run-time is supported and work is proceeding on dynamic analysis.

μC++ [27] is somewhat similar to Concurrent C++ [46] but provides much more extensive facilities and has given more thought in the design and solution of potential problems. Three basic properties of execution are considered; *threads* which define independent execution of code, *execution state* which defines the information needed for concurrent execution and *mutual exclusion* which controls access to shared resources. All eight combinations of the presence or absence of these properties are analysed and potentially hazardous ones rejected, for instance it does not make sense for an object to have a thread but no execution state. This analysis suggested three new constructs; *coroutines* which are objects with their own state, *monitors* which are objects with mutual exclusion and *tasks* which have a state and a thread of control. Inheritance is implemented within these constructs but there are a few problems with some combinations of function types.

Mentat [49, 50] is the combination of a C++ based language called MPL and a run-time system. The unit of computation is the class member function, the keyword *mentat* informs the system of the potential for parallelism and persistence can also be specified for member functions. The emphasis in the design of Mentat is on high performance and ease of writing programmes, although code reuse and parallel encapsulation are also taken into account. The basic model seems to be actors with ideas borrowed from dataflow architectures with an enhanced RPC style implementation. One of the most impressive aspects of Mentat if the number of parallel architectures to which it has been ported, including Sun 4s, Intel iPSC/2, Intel iPSC/860, Silicon Graphics Iris, IBM RS 6000, TMC CM-5 and Intel Paragon. Performance figures for a biological application also look good [51] although there is a lot of inherent parallelism in the application.

pSather [44] is a parallel version of Sather [96] which is a reduced and optimised language version of the object-oriented language Eiffel [43]. pSather is based on concurrent threads and monitors and there is an optional lock status which causes calling threads to block until the monitor is unlocked although there is a *try* statement to circumvent this. This language was originally written for shared memory machines but was extended to distributed-memory machines with the provision of a clustering mechanism in the implementation of a global address space. There is also provision for moving and copying objects.

Eiffel has itself been a subject for concurrent implementation based on communicating objects, [68] describes a concurrency library as an add on to the basic Eiffel system. This uses asynchronous message passing with futures and proxies. Eiffel// [31] also uses asynchronous communication but extends the core Eiffel language.

CHARM++ [67] is one of the most mature C++ based concurrent languages in that it has been ported to commercial parallel machines and implements most of the C++ object-oriented features. Threads of execution are called *chares* in CHARM terminology and objects are classified as *sequential*, *concurrent*, *replicated*, *shared* and *communication* allowing abstraction to occur over parallel structure, communications patterns and object access. Replicated objects are for controlling data parallel algorithms and having separate sequential and concurrent objects means that remote object access is not transparent. Communications are asynchronous and futures are implemented.

Most of the C++ features such as inheritance, dynamic binding and overloading are implemented for parallel as well as sequential objects although some of the decision making is done at run-time rather than compile-time. Different types of shared objects are defined such as *read-only*, *write-once* and *monotonic* objects and there are a number of dynamic load balancing strategies available such as *random* allocation of tasks to processors and local optimisation within a neighbourhood. A number of support tools are available such as performance analysis and a visual editor for synchronisation constraints.

CC++ (Compositional C++) [33] is an ambitious project with wide-ranging goals including providing a unified framework for different types of systems such as declarative and object-oriented systems, deterministic and non-deterministic programmes and various modes of computation such as distributed systems and user-interface systems. CC++ is also intended for rapid prototyping and includes declarative extensions to facilitate composition of existing programmes and can also be used in correctness proofs of applications.

CC++ borrows ideas from concurrent logic languages and from dataflow languages and is RPC based. The host language is C++, extended by six constructs; *parallel block*, *spawn*, *atomicity*, *logical processors* (classes with no global or static variables), *global pointers* and *synchronisation* of variables. Most of the standard object-oriented features are implemented but there are restrictions, for instance, there are no *sync unions*.

Concurrent/Distributed Smalltalk [45, 92], since Smalltalk [48] was the first truly object-oriented language it is not surprising that it has become the basis for many concurrent and distributed implementations. *ConcurrentSmalltalk* [41] unifies objects and processes and both synchronous and asynchronous communications are provided, although RPC methods are sometimes preferred if compatibility with sequential Smalltalk is required [25]. Objects are also distributed and can concurrently service messages. *DistributedSmalltalk* is intended to provide a sharing mechanism for distributed objects, [42] describes a system which uses *proxies* and object migration in conjunction with distributed garbage collection. Smalltalk is, however, compatible with other modes of computation, for instance [25] describes an actor-based Smalltalk implementation called *Actalk* which implements most of the facilities provided by ABCL/1. Concurrent and Distributed Smalltalk have also been combined in a single system [91].

DOWL [1] is a distributed version of the Trellis/Owl system [95, 69]. This uses a combination of RPC and proxies to maintain semantic compatibility with the Trellis system. Object placement is generally transparent, immutable objects can migrate but mutable objects are fixed with proxies created remotely, but can be overridden by explicit object location control with the *$fixed*, *$mobile* and *$locate* constructs. Objects can also be linked with the *$attach* construct and *$replicate* and *$visit* are provided, allowing replication and temporary relocation.

Orca [17, 16] is a strongly-typed language similar to Modula-2 with explicit control over parallelism and data location. The programmer sees a set of distributed abstract data types with a lock mechanism for mutual exclusion and a *condition synchronisation* mechanism implemented as boolean *guards*. Inheritance is not implemented but abstract data types can be nested. A set of standard data types is built into the language, including graphs, which allows more efficient run-time implementation of such types. The communication model is based on shared data but there is support for replication and migration of objects in a mostly transparent manner.

Ellie [11] is an object-oriented language aimed at exploiting fine-grained parallelism in MIMD computers, its current implementation is aimed at meshes of transputers but machine independence is one of its design goals. Communication is by RPC with futures, similar to ABCL/1. A novel feature of the compiler is that it starts with fine-grained parallelism (at the level of multiplication of integers, for instance) and then attempts to predict the optimal granularity for a given architecture by coalescing fine-grained into coarser-grained parallelism using communications constraints, although this has not yet been implemented automatically.

ABC++ (Active Base Class) [14] was designed with the intention of adding concurrency to C++ without placing the burden of explicit concurrency and synchronisation control on the programmer. Communications can be synchronous or asynchronous and is based on blocking, non-blocking and future RPC modelling method invocation. Active objects are supported along with a selective method acceptance mechanism, messages are also queued and since only one thread exists per object it is claimed this makes object integrity automatic. A form of proxies are also provided for remote object access. The implementation of C++ is fairly complete but there are a few limitations such as friend classes only being able to access virtual member functions and no public or static data class members.

PARC++ [110] attempts to combine shared memory and distributed memory programming in a single system aimed at providing some degree of architectural independence. The basic model of execution is similar to PRESTO where the sequential/parallel execution of an object is decided by the implementor rather than the programmer. Two locking mechanisms are provided; *blocklock* which attempts to lock and then fails and *spinlock* which keeps trying until the lock is available, *monitors* are also provided. Message passing is provided by mailboxes and can be synchronous or asynchronous. PARC++ also provides interfaces to parallel visualisation and debugging tools, for instance ParaGraph [54].

The above list of languages is not complete, there have been many other attempts to introduce concurrency into an object-oriented language or visa versa, for instance Jade [73] allows parallelisation of sequential programmes, Alba [55], an actor based implementation language for a database machine, cooC [113] which is based on Objective-C, P++ [78] gives parallel implementations of array operations for C++, Cantor [15] is an experimental fine-grained actor-based language, Procol [117] uses a constrained protocol message passing system, C++ for transputers [116] presents a translation system from C++ to transputer code and SINA [115] which supports concurrency within objects.

An important class of languages are those designed for data-parallel programming. Some of these are object-oriented, for instance C** [76], a development of C* [98], is an object-oriented language designed to exploit *large-grain data parallelism* and is based on Chien's *concurrent aggregates* [35]. These languages overlap with MIMD languages, however, Hatcher and Quinn showed that data-parallel code could be compiled for both message-passing and shared-memory computers, and developed Dataparallel C [53] which has subsequently been enhanced to support multiple data-parallel modules [100].

In addition to the languages mentioned above, there are also a number of systems which support concurrent object-oriented programming, some of them including languages themselves which are intended as vehicles for writing higher-order languages:

Rosette [111] is an actor based operating system designed to support dynamic problems with variable grain size, dynamic resource management and reflection. There are two principal components; an *interface layer* and a *system environment*. Rosette also includes a concurrent object-oriented language that is designed to be translated into a host language, extending it with high-level abstraction facilities.

VOOM [18] is a parallel virtual object-oriented machine intended to address problems of portability in object-oriented languages, both sequential and parallel. The main components are an *execution unit*, a *pre-fetch unit*, a *memory-management unit* and a *communication unit*.

Choices (class hierarchical open interface for custom embedded systems) [29] is an object-oriented operating system developed in C++. The emphasis is on the use of prototyping and both inheritance and polymorphism are supported. Runtime support includes garbage collection and object classes which can be dynamically created.

Amadeus [105] is designed to support parallel programming of distributed, heterogeneous systems. Objects are passive, persistent and distributed and there is support for clustering (keeping related objects together) and load balancing. Currently, two languages are implemented, C** (an enhanced C++) and Eiffel**.

DOCASE [90] is a system for designing distributed applications and comprises two languages; DODL a design language and T^SL (superimposition language) which defines object migration. There is also an overall design assistant called DOCASE.

COOL (Chorus object-oriented layer) [77] is an operating system whose design goals are to alleviate the mismatch between the abstraction mechanisms required by object-oriented programming languages and those provided by the operating system. COOL can support distributed shared memory and message passing and is based on threads, programming is in a dedicated language called COOL [85] influenced by Pascal and Modula-2.

An important class of concurrent languages not mentioned here are concurrent logic languages [109], some of which are object-oriented. Examples of these are; the OBJ series of languages [47] and I+ [94].

3.1 General principles from COOP languages

None of the reviewed languages implement all of the features that have come to be expected in modern sequential object-oriented languages. CHARM++ perhaps comes closest but even this language imposes restrictions on the programmer with respect to certain combinations of features. It cannot be said, yet, that an arbitrary programme written in an object-oriented language can be automatically implemented in parallel. Most of the languages, in fact, leave practically all the decisions concerning parallelism to the programmer, although some attempt to provide automatic support for these activities, for instance, Ellie.

A surprising number of them are object-based rather than object-oriented, particularly the early attempts such as Concurrent C++, either because omitting inheritance leads to an easier implementation or the recognised problems associated with inheritance were not addressed. There is no doubt that it is easier to implement an object-based language concurrently but object-oriented languages lose much of their expressive power without inheritance, particularly with respect to code reuse.

The abstraction mechanisms provided by object-oriented languages can be applied to various aspects of parallel code design and implementation, as well as the more usual data and code abstraction. Examples are the communications strategy in Clix, the execution behaviour in PRESTO and concurrency control and synchronisation in ABC++. Some systems even try to unify multiple programming entities with a single abstraction mechanism, for instance BETA. There is no consensus as to how to apply these abstraction mechanisms in a concurrent object-oriented language but this is perhaps one of the most promising aspects of COOP technology, the potential to package up communications and computation patterns, somewhat similar to the algorithmic skeletons [37] concept that has been applied to parallelising functional languages.

Indeed some languages try to introduce a degree of declarativeness into the host language, for instance CC++ and A'UM. There is a slight problem here in that the power of object-oriented programming derives in part from the use of side-effects in objects, objects exist and have a state which persists and changes over time in the real world and very often it is real-world objects that are being modelled in an object-oriented programme. This is particularly true in concurrent object-oriented programming where very often these objects are realised as physically separate entities, and is a similar problem to the question of whether a functional language should be made object-oriented [108].

There is no convincing arguments as to whether synchronous or asynchronous communications are better for a particular application, many recent systems simply allow both modes, in conjunction with other options, for instance RPC versus message passing. Since both modes can emulate each other the decision on which should be used in a particular application or language may depend on considerations such as ease of programming, reliability of code (synchronous communications are generally regarded as being less error prone than asynchronous) and also the nature of the target hardware since hardware support for communications can likewise be classified into synchronous and asynchronous.

3.2 Features provided by concurrent object-oriented languages

Few languages provide the full range of object-oriented techniques, most of them are object-based rather than object-oriented and thus lack features such as inheritance and dynamic binding. Indeed inheritance seems to be one of the most difficult features of an object-oriented language to implement fully and reliably in a parallel context. All the languages that attempt to implement inheritance place some kind of restriction on the nature of classes or variables during the inheritance process for instance CHARM++ and SR. Similar problems are reported for other object-oriented features such as multiple inheritance (which complicates consistency maintenance), dynamic binding and overloading.

As far as support for parallel programming is concerned, a number of languages attempt to provide some form of dynamic load balancing (for instance, CHARM++, Amadeus and Emerald) although there is little agreement upon whether compile-time or run-time load balancing is best, for instance Amber's object placement is under programmer control whereas Emerald's is done at run-time. The reuse of parallel objects is almost guaranteed by the use of an object-oriented parallel system but some systems make more of an effort to facilitate this than others, for example PRESTO and Mentat. Dynamic load balancing is potentially a fruitful avenue of research, particularly with the advent of methods such as computational reflection [82] on which some languages are starting to be based [121].

3.3 Parallel Hardware Supported by COOP Languages

A bewildering array of parallel hardware has been targetted by designers of concurrent object-oriented languages. Some systems have actually been designed with these types of languages in mind, for instance the DOOM architecture. Another point is that there are also virtual machines which have been developed with concurrent object-oriented languages in mind, for instance VOOM.

Shared memory machines are very popular for COOP languages, presumably because some of the burden of communications efficiency is the responsibility of the hardware designer. Systems such as the Sequent Symmetry (CHARM++, PRESTO, pSather, C++ Parmacs, C**) and Sequent Balance (PRESTO, C++ Parmacs) have been used.

Distributed memory machines such as the nCUBE/2 (CHARM++), EDS (European Declarative System) (PARC++), Transputer (Ellie) and CM-5 (CHARM++, pC++, Olden, Mentat, pSather) form the most popular class of machines for COOP implementation. This is probably due to the perception of objects as physically discrete and persistent entities realised on distinct processors with local resources. It does appear that this is the most natural means of implementing a concurrent object-oriented language but many problems still remain to be solved such as automatically mapping processes to a network of processors.

Distributed systems such as networks of IBM PCs (PARC++), CHORUS MiX (PARC++) and DEC MicroVAX II workstations (Emerald) also feature as suitable platforms. An interesting possibility here is the use of heterogeneous networks (Mentat).

There are also a number of other systems which have been used as platforms for concurrent object-oriented technology; Fujitsu AP1000 (ABCL/1), EM-4 (ABCL), J-machine (CST), iPSC/860 (Olden, Mentat), iPSC/2 (Mentat), Silicon Graphics Iris (Mentat), IBM RS 6000 (Mentat, ABC++) and Intel Paragon (Mentat). These span a wide range of architectural features and possibly indicate a high degree of generality in the COOP approach.

3.4 Summary

Some foundational issues in concurrent object-oriented languages were reviewed. The implicit assumption in all of the languages that were mentioned was briefly justified (that object-orientation is a useful way of structuring programmes for parallel execution, if not the fundamental source of parallelism). There are a number of technical issues that have to be addressed before deciding upon a concurrent object-oriented paradigm and these were also reviewed.

A large number of languages were then each briefly discussed which cover a wide range of paradigms, from lambda calculus to stream semantics but most of them use the object-based idea whereby objects are discrete entities in the parallel implementation. There has been steady progress over the years with problems being identified and addressed and new ideas being tried out. Some languages are now becoming fully fledged object-oriented languages with most of the features that have come to be expected of a modern object-oriented language but there are still many restrictions and limitations.

One of the greatest outstanding problems is associated with inheritance and has come to be known as the *inheritance anomaly* for which several solutions have been proposed [86]. One recent development is that of computational reflection [121] on which some languages are based.

References

[1] B. Achauer. The DOWL Distributed Object-Oriented Language. *CACM*, 36(9):48–55, Sep 1993.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] G. Agha. Foundational Issues in Concurrent Computing. *SIGPLAN Notices*, 24(4):60–66, Apr 1989.

[4] G. Agha. Concurrent Object-Oriented Programming. *CACM*, 33(9):125–141, Sep 1990.

[5] G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. *SIGPLAN Notices*, 28(7):23–32, Jul 1993.

[6] G. Agha, S. Frolund, W. Y. Kim, R. Panwar, A. Patterson, and D. Sturman. *Abstraction and Modularity Mechanisms for Concurrent Computing, In [9]*, chapter 1, pages 3–21. MIT Press, 1993.

[7] G. Agha and C. Hewitt. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming, In [102]*, pages 49–74. MIT Press, 1987.

[8] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computations. In *Third International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, Aug 1992.

[9] G. Agha, P.Wegner, and A. Yonezawa. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[10] P. America. *Inheritance and Subtyping in a Parallel Object-Oriented Language,* In [21], volume 276 of *LNCS*, pages 234–242. Springer-Verlag, 1987.

[11] B. Andersen. A General, Fine-Grained, Machine Independent, Object-Oriented Language. *ACM SIGPLAN Notices*, 29(5):17–35, May 1994.

[12] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An Overview of the SR Language and Implementation. *ACM TOPLAS*, 10(1):51–86, Jan 1988.

[13] J. K. Annot and P. A. M. den Haan. *POOL and DOOM: The Object-Oriented Approach,* In [114], chapter 3, pages 47–79. Wiley, 1990.

[14] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G. R. Gao. ABC++: Concurrency by Inheritance in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[15] W. Athas and N. Boden. Cantor: An Actor Programming System for Scientific Computation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Proceedings of the NSF Workshop on Object-Based Concurrent Programming*, SIGPLAN Notices, pages 66–68. ACM, Apr 1989.

[16] H. E. Bal and M. Frans Kaashoek. Object Distribution in Orca using Compile-time and Run-time Techniques. *ACM SIGPLAN Notices*, 28(10):162–177, Oct 1993.

[17] H. E. Bal, M. Frans Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. Software Engineering*, 18(3):190–205, Mar 1992.

[18] A. T. Balou and A. N. Refenes. The Design and Implementation of VOOM: A Parallel Virtual Object-Oriented Machine. *Microprocessing and Microprogramming*, 32(1-5):289–296, 1991.

[19] B. Beck. Shared Memory Parallel Programming in C++. *IEEE Software*, pages 38–48, Jul 1990.

[20] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software - Practice and Experience*, 18(8):713–732, Aug 1988.

[21] J. Bezivin, J.-M. Hullot, P. Cointe, and H. Lieberman. *ECOOP '87 European Conference on Object-Oriented Programming*, volume 276 of *LNCS*. Springer-Verlag, 1987.

[22] A. Black, N. Hutchison, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Trans. Software Engineering*, SE-13(1):65–76, 1987.

[23] S. H. Bokhari. On the Mapping Problem. *IEEE Trans. on Computers*, C-30(3):207–214, 1981.

[24] T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[25] J.-P. Briot. From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80. *SIGPLAN Notices*, 24(4):69–72, Apr 1989.

[26] J.-P. Briot and J. de Ratuld. Design of a Distributed Implementation of ABCL/1. *SIGPLAN Notices*, 24(4):15–17, Apr 1989.

[27] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Young, and C. R. Zarnke. μC++: Concurrency in the Object-oriented Language C++. *Software - Practice and Experience*, 22(2):137–172, Feb 1992.

[28] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley, Reading, MA, 1990.

[29] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing *Choices*: An Object-Oriented System in C++. *CACM*, 36(9):117–126, 1993.

[30] D. Caromel. A General Model for Concurrent and Distributed Object-Oriented Programming. *SIGPLAN Notices*, 24(4):102–104, Apr 1989.

[31] D. Caromel. Toward a Method of Object-Oriented Concurrent Programming. *CACM*, 36(9):90–102, Sep 1993.

[32] N. Carriero and D. Gelertner. Linda in Context. *CACM*, 32(4):444–459, Apr 1989.

[33] K. Mani Chandy and C. Kesselman. *CC++: A Declarative Concurrent Object-Oriented Programming Notation, In [9]*, chapter 11, pages 281–313. MIT Press, 1993.

[34] J. S. Chase, F. G. Amadar, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, volume 23 of *ACM SIGOPS*, pages 147–158, Dec 1989.

[35] A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[36] T. W. Christopher. Message Driven Computing and its Relationship to Actors. *SIGPLAN Notices*, 24(4):76–78, Apr 1989.

[37] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, 1989.

[38] B. J. Cox and A. J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 2nd edn. edition, 1991.

[39] O.-J. Dahl, B. Myrhaug, and K. Nygard. *SIMULA 67 Common Base Language.* Norwegian Computer Centre, Oslo, 1968.

[40] W. Dally. *The J-machine: System Support for Actors, In [57].* MIT Press, 1990.

[41] W. J. Dally and A. A. Chien. Object-Oriented Concurrent Programming in CST. *SIGPLAN Notices*, 24(4):28–31, Apr 1989.

[42] D. Decouchant. *A Distributed Object Manager for the Smalltalk-80 System,* In [70], chapter 19, pages 487–520. ACM Press, 1989.

[43] U. Eshkar and S. Schleimer. Eiffel and C++. *Journal of Object-Oriented Programme*, 7(7):8–10, 1994.

[44] J. A. Feldman, C.-C. Lim, and T. Rauber. The Shared-Memory Language pSather on a Distributed-Memory Multiprocessor. *SIGPLAN Notices*, 28(1):17–20, Jan 1993.

[45] Y. Gao and C. K. Yuen. A Survey of Implementations of Concurrent, Parallel and Distributed Smalltalk. *ACM SIGPLAN Notices*, 28(9):29–35, Sep 1993.

[46] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software - Practice and Experience*, 18(12):1157–1177, 1988.

[47] J. Goguen and J. Meseguer. *Unifying functional, object-oriented and relational programming with logical semantics,* In [102], pages 417–477. MIT Press, Cambridge, Mass., 1987.

[48] A. Goldberg and D. Robson. *Smalltalk80: the language and its implementation.* Addison-Wesley, Reading, Mass., 1983.

[49] A. S. Grimshaw. Easy-to-use Object-Oriented Parallel Processing with Mentat. *Computer*, 26(5):39–51, 1993.

[50] A. S. Grimshaw. The Mentat Computation Model Data-Driven Support for Object-Oriented Parallel Processing. Technical Report CS-93-30, Department of Computer Science, University of Virginia, May 1993.

[51] A. S. Grimshaw, E. A. West, and W. R. Pearson. No Pain and Gain - Experiences with Mentat on a Biological Application. *Concurrency - Practice and Experience*, 5(4):309–328, 1993.

[52] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7:501–538, 1985.

[53] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[54] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, 1991.

[55] J. Hernandez, P. de Miguel, M. Barrena, J. M. Martinez, A. Polo, and M. Nieto. ALBA: A Parallel Language Based on Actors. *SIGPLAN Notices*, 28(4):11–20, Apr 1993.

[56] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[57] C. Hewitt and G. Agha. *Towards Open Information Systems Science.* MIT Press, 1990.

[58] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[59] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *CACM*, 17(10):549–557, Oct 1974.

[60] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[61] C. Houck and G. Agha. HAL: A High-level Actor Language and its Distributed Implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, Aug 1992.

[62] J. H. Hur and K. Chon. *Overview of a Parallel Object-Oriented Language CLIX,* In [21], volume 276 of *LNCS*, pages 265–273. Springer-Verlag, 1987.

[63] D. Jefferson. Virtual Time. *ACM TOPLAS*, 7:404–425, 1985.

[64] D. Jordan. Implementation Benefits of C++ Language Mechanisms. *CACM*, 33(9):61–64, Sep 1990.

[65] E. Jul, H. Levy, N. Hutchison, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, Feb 1988.

[66] D. Kafura. Concurrent Object-Oriented Real-Time Systems Research. *SIGPLAN Notices*, 24(4):203–205, Apr 1989.

[67] L. V. Kale and S. Krishnan. CHARM++: A Portable Object-Oriented System Based on C++. *ACM SIGPLAN Notices*, 28(10):91–108, Oct 1993.

[68] M. Karaorman and J. Bruno. Introducing Concurrency to a Sequential Language. *CACM*, 36(9):103–116, Sep 1993.

[69] M. Kilian. Trellis: Turning Designs into Programs. *CACM*, 33(9):65–67, Sep 1990.

[70] W. Kim and F. H. Lochovsky. *Object-Oriented Concepts, Databases and Applications.* Addison-Wesley, 1989.

[71] T. Korson and J. D. McGregor. Understanding Object-Oriented: A Unifying Paradigm. *CACM*, 33(9):40–60, Sep 1990.

[72] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *The BETA Programming Language,* In [102], pages 7–48. MIT Press, 1987.

[73] M. S. Lam and M. C. Rinard. Coarse-Grain Parallel Programming in Jade. *SIGPLAN Notices*, 26(7):94–105, Jul 1991.

[74] P. J. Landin. A Correspondence between Algol 60 and Church's Lambda Notation: Part I. *CACM*, 8(2):89–101, 1965.

[75] G. Lapalme and P. Salle. Plasma-II: An Actor Approach to Concurrent Programming. *SIGPLAN Notices*, 24(4):81–83, Apr 1989.

[76] J. R. Larus, B. Richards, and G. Viswanathan. C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language. Technical Report UW Technical Report #1126, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, Nov 1992.

[77] R. Lea, C. Jacquemot, and E. Pillevesse. COOL: System Support for Distributed Programming. *CACM*, 36(9):37–46, Sep 1993.

[78] M. Lemke and D. Quinlan. P++, A Parallel C++ Array Class Library for Architecture-Independent Development of Structured Grid Applications. *SIGPLAN Notices*, 28(1):21–23, Jan 1993.

[79] H. Liberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In *Proceedings ACM Conf. on OOPSLA '86*, pages 214–223, Sep 1986.

[80] J. Lim and R. E. Johnson. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices*, 24(4):165–167, Apr 1989.

[81] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices, OOPSLA '87*, 1987.

[82] P. Maes. *Computational Reflection*. PhD thesis, Vrije University, Brussels, Belgium, 1987.

[83] G. Manning. A Peak at Acore, an Actor Core Language. *SIGPLAN Notices*, 24(4):84–86, Apr 1989.

[84] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation*. Prentice-Hall, 1995.

[85] K. Maruyama and N. Raguideau. Concurrent Object-Oriented Language "COOL". *ACM SIGPLAN Notices*, 29(9):105–114, Sep 1994.

[86] S. Matsuoka and A. Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, In [9]*, chapter 4, pages 107–150. MIT Press, 1993.

[87] P. Mehrotra and J. van Rosendale. Concurrent Object Access in Blaze 2. *SIGPLAN Notices*, 24(4):40–42, Apr 1989.

[88] G. J. Michaelson and N. R. Scaife. Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, 5(3):345–382, Jul 1995.

[89] G. F. Mota, M. L. Nelson, and U. R. Kodres. Object-Oriented Decomposition for Distributed Systems. *Microprocessing and Microprogramming*, 40:91–102, 1994.

[90] M. Muhlhauser, W. Gerteis, and L. Heuser. DOCASE: A Methodic Approach to Distributed Programming. *CACM*, 36(9):127–138, Sep 1993.

[91] T. Nakajima, Y. Yokote, S. Ochiai, and T. Nagamatsu. Distributed Concurrent Smalltalk, A Language and System for the Interpersonal Environment. *SIGPLAN Notices*, 24(4):43–45, Apr 1989.

[92] M. L. Nelson. Concurrency & Object-Oriented Programming. *SIGPLAN Notices*, 26(10):63–72, Oct 1991.

[93] M. L. Nelson. Considerations in Choosing a Concurrent/Distributed Object-oriented Programming Language. *ACM SIGPLAN Notices*, 29(12):66–71, Dec 1994.

[94] K. W. Ng and C. K. Luk. I+ - A Multiparadigm Language for Object-Oriented Declarative Programming. *Computer Languages*, 21(2):81–100, 1995.

[95] P. O'Brien, D. Halbert, and M. Killian. The Trellis Programming Environment. *SIGPLAN Notices*, pages 91–102, 1987.

[96] S. M. Omohundro. The Sather Programming Language. *Dr. Dobbs Journal*, 18(10):68, 1993.

[97] J. Padget, R. Bradford, and J. Fitch. Concurrent Object-Oriented Programming in Lisp. *Comp. J.*, 34(4):311–319, 1991.

[98] J. R. Rose and G. L. Steele. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, pages 2–16, Santa Clara, California, May 1987.

[99] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Prototyping Parallel Algorithms using Standard ML. In D. Pycock, editor, *Proceedings of the 6th British Machine Vision Conference*, volume 2, pages 671–680, University of Birmingham, Birmingham, Sep 1995.

[100] B. K. Seevers, M. J. Quinn, and P. J. Hatcher. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. *International Journal of Parallel Programming*, 21(5):363–386, 1992.

[101] S. K. Shrivastava and F. Panzieri. The Design of a Reliable Remote Procedure Call Machanism. *IEEE Trans. Computers*, 31(7):692–697, Jul 1982.

[102] B. Shriver and P. Wegner. *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[103] A. Z. Spector. Perform Remote Operations Efficiently on a Local Computer Network. *CACM*, 25(4):246–260, Apr 1982.

[104] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edn. edition, 1993.

[105] B. Tangney, A. Condon, V. Cahill, and N. Harris. Requirements for Parallel Programming in Object-oriented Distributed Systems. *Comp. J.*, 37(6):499–508, 1994.

[106] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *ACM Operating Systems Review*, 24(3):68–79, Jul 1990.

[107] D. A. Thomas, W. R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, and B. Berry. Actra: A Multitasking/Multiprocessing Smalltalk. *SIGPLAN Notices*, 24(4):87–90, Apr 1989.

[108] L. Thorup and M. Tofte. Object-Oriented Programming and Standard ML. In *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 41–49. Inria Report No. 2265, Jun 1994.

[109] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 23(2):89–123, 1995.

[110] K. Todter, C. Hammer, and W. Struckmann. PARC++: A Parallel C++. *Software - Practice and Experience*, 25(6):623–636, Jun 1995.

[111] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An Object-Oriented Concurrent Systems Architecture. *SIGPLAN Notices*, 24(4):91–93, Apr 1989.

[112] C. Tomlinson and M. Scheevel. *Concurrent Object-Oriented Programming Languages,* In [70], chapter 5, pages 79–126. ACM Press/Addison-Wesley, 1989.

[113] R. Trehan, N. Sawashima, A. Morishita, I. Tomoda, T. Imai, and K.-I. Maeda. Concurrent Object Oriented 'C' (cooC). *SIGPLAN Notices*, 28(2):45–52, Feb 1993.

[114] P. C. Treleaven. *Parallel Computers Object-Oriented, Functional, Logic.* Series in Parallel Computing. Wiley, 1990.

[115] A. Tripathi, E. Berge, and M. Aksit. An Implementation of the Object-Oriented Concurrent Programming Language SINA. *Software - Practice and Experience*, 19(3):235–256, Mar 1989.

[116] T. Ungerer. Parallelising C++-Programs for Transputer Systems. *Microprocessing and Microprogramming*, 32(1-5):463–470, 1991.

[117] J. van den Bos. PROCOL: A Protocol-Constrained Concurrent Object-Oriented Language. *SIGPLAN Notices*, 24(4):149–151, Apr 1989.

[118] A. M. Wallace, G. J. Michaelson, P. McAndrew, K. Waugh, and W. Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-level Vision. *IEEE Computer*, 25(2):43–53, Feb 1992.

[119] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge, Mass., 1990.

[120] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura. Efficient Implementations of Concurrent Object-Oriented Languages on Multicomputers. *IEEE Parallel and Distributed Technology*, (to be published), 1995.

[121] A. Yonezawa and T. Watanabe. An Introduction to Object-Based Reflective Concurrent Computation. *SIGPLAN Notices*, 24(4):50–54, Apr 1989.

[122] K. Yoshida and T. Chikayama. A'UM = Stream + Object + Relation. *SIGPLAN Notices*, 24(4):55–58, Apr 1989.

[123] K. Yoshida and T. Chikayama. A'UM - A Stream-Based Concurrent Object-Oriented Language. *New Generation Computing*, 7:127–157, 1990.